AFRL-RI-RS-TR-2012-301

SELECTIVE, EMBEDDED, JUST-IN-TIME SPECIALIZATION (SEJITS): PORTABLE PARALLEL PERFORMANCE FROM SEQUENTIAL, PRODUCTIVE, EMBEDDED DOMAIN-SPECIFIC LANGUAGES

UNIVERSITY OF CALIFORNIA, BERKELEY

DECEMBER 2012

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the 88th ABW, Wright-Patterson AFB Public Affairs Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2012-301 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

FOR THE DIRECTOR:

/ S /

CHRISTOPHER FLYNN
Work Unit Manager

/ S /

RICHARD MICHALAK
Acting Tech Advisor, Computing & Communications Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
DECEMBER 2012	FINAL TECHNICAL REPORT	JUN 2010 – JUN 2012

4. TITLE AND SUBTITLE

Selective, Embedded, Just-In-Time Specialization (SEJITS): Portable Parallel Performance from Sequential, Productive, Embedded Domain-Specific Languages

5a. CONTRACT NUMBER
FA8750-10-1-0191

5b. GRANT NUMBER
N/A

5c. PROGRAM ELEMENT NUMBER
61101E

6. AUTHOR(S)

Armando Fox

5d. PROJECT NUMBER
BG69

5e. TASK NUMBER
BK

5f. WORK UNIT NUMBER
LY

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California, Berkeley
2150 Shattuck Avenue, RM 300
Berkeley, CA 94704-5940

8. PERFORMING ORGANIZATION REPORT NUMBER
N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RITA
525 Brooks Road
Rome NY 13441-4505

10. SPONSOR/MONITOR'S ACRONYM(S)
N/A

11. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFRL-RI-RS-TR-2012-301

12. DISTRIBUTION AVAILABILITY STATEMENT
Approved for Public Release; Distribution Unlimited. PA# 88ABW-2012-6474
Date Cleared: 12 DEC 2012

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Domain-expert productivity programmers desire scalable application performance, but usually must rely on efficiency programmers who are experts in explicit parallel programming to achieve it. Since such efficiency programmers are rare, to maximize reuse of their work we propose SEJITS (Selective Embedded Just-in-Time Specialization), a methodology for encapsulating efficiency programmers' strategies in mini-compilers for domain-specific embedded languages (DSELs). The DSELs are then glued together by embedding in a common high-level host language familiar to productivity programmers. With a variety of implemented examples, including structured grids, large-scale graph analysis, audio processing, and communication-avoiding linear algebra, we demonstrate that our approach allows productivity programmers' code to approach the performance of efficiency-level code while retaining the ease of authoring and maintenance provided by productivity languages.

15. SUBJECT TERMS
SEJITS, Domain-Specific Embedded Languages, Linear Algebra, Productivity Programmers

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
					CHRISTOPHER FLYNN
a. REPORT	b. ABSTRACT	c. THIS PAGE	UU	33	19b. TELEPONE NUMBER (*Include area code*)
U	U	U			**N/A**

Contents

List of Figures

List of Tables

1 Summary

Domain-expert *productivity programmers* desire scalable application performance, but usually must rely on *efficiency programmers* who are experts in explicit parallel programming to achieve it. Since such efficiency programmers are rare, to maximize reuse of their work we propose encapsulating their strategies in mini-compilers for domain-specific embedded languages (DSELs) glued together by a common high-level host language familiar to productivity programmers. Our approach is unique in two ways. First, each mini-compiler not only performs conventional compiler transformations and optimizations, but includes imperative procedural code that captures an efficiency expert's strategy for mapping a narrow domain onto a specific type of hardware. Second, the mini-compilers emit source code in an efficiency language such as C++/OpenMP or CUDA [19] that allows targeting low-level hardware optimizations using downstream compilers combined with auto-tuning, where many candidate implementations are generated and run to discover the best-performing variant. The result is source- and performance-portability for productivity programmers, and performance that rivals that of hand-coded efficiency-language implementations.

SEJITS is a DSEL construction methodology that supports capturing expert programmers' strategies with a combination of code templates and tree-driven code transformations based on a high-level, platform-independent representation of a problem. Our approach provides benefits that are difficult to achieve with libraries or general-purpose languages, including performance portability across platform types (e.g. CPU vs. GPU), tuning of inlined higher-order functions, just-in-time autotuning based on specific problem inputs, and optimizations based on narrow domain knowledge that is hard to capture in a general-purpose language. Our DSELs are as easy to use as libraries for domain experts and not much harder to create for efficiency programmers. We describe a framework that supports the SEJITS methodology and four implemented DSELs supporting common computation kernels. The nontrivial applications that use these kernels achieve performance portability across platforms and achieve measured performance comparable to hand-coded efficiency implementations, despite being written entirely in the productivity language Python (http://python.org).

In addition, we integrate SEJITS with the Knowledge Discovery Toolkit (KDT), a framework that supports complex analytic queries on massive semantic graphs. Specifically, we show two ways to customize KDT using SEJITS. First, the user can write custom graph algorithms by specifying operations between edges and vertices. These programmer-specified operations are called *semiring operations* due to KDT's linear-algebraic abstractions. Second, the user can customize existing graph algorithms by writing filters that pass only a subset of individual vertices and edges of interest to the algorithmic kernel. For high productivity, both semiring ops and filters can be written in Python, using SEJITS to automatically translate them to a lower-level efficiency language. We show that the performance obtained is comparable to that of the native Combinatorial BLAS (Basic Linear Algebra Subprograms) engine (which is written in a low-level productivity language), and therefore that the productivity gained by using a high-level filtering language incurs essentially no cost in performance. Indeed, we also present a new roofline model that estimates the best possible performance for communication-bound graph traversals, and show that our implementations do not significantly deviate from that roofline.

Overall, our results demonstrate that for several interesting classes of problems, efficiency-level parallel performance can be achieved by packaging efficiency programmers' expertise in a reusable framework that is easy to use for both productivity programmers and efficiency programmers.

The authors are grateful for feedback from Ras Bodik, Hasan Chafi, James Demmel, Joseph Hellerstein, Michael Mccool, and Tim Mattson.

This work was performed at the UC Berkeley Parallel Computing Laboratory (Par Lab), supported by DARPA (contract #FA8750-10-1-0191) and by the Universal Parallel Computing Research Centers (UPCRC) awards from Microsoft Corp. (Award #024263) and Intel Corp. (Award #024894), with matching funds from the UC Discovery Grant (#DIG07-10227) and additional support from Par Lab affiliates National Instruments, NEC, Nokia, NVIDIA, Oracle, and Samsung. This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05-CH-11231. Authors from Lawrence Berkeley National Laboratory were supported by the DOE Office of Advanced Scientific Computing Research under contract number DE-AC02-05-CH-11231. The authors from UC Santa Barbara were supported in part by NSF grant CNS-0709385, by a contract from Intel Corporation, by a gift from Microsoft Corporation and by the Center for Scientific Computing at UCSB under NSF Grant CNS-0960316.

2 Introduction

Our goal is best illustrated by a short scenario that is common among our scientific colleagues. Paul, a productivity programmer whose main research area is biology, has prototyped a new graph algorithm in Python, a language that supports his domain well because of its library support for reading molecule files, graphing results, and so on. Since Python is too slow to run his algorithm on non-toy problems, Paul retains Elena, an efficiency programmer, to create a high-performance version of his algorithm that can exploit the parallelism of Paul's multicore servers with GPUs.

Since programmers like Elena are rare, we would like to reuse their work as widely as possible. Rather than rewriting Paul's application entirely in C++ or CUDA, Elena might "package" the kernel of his algorithm as a library that can be called from Python, though the library would have to include enough generality to allow other scientists to tweak the algorithm without being efficiency programmers themselves. Furthermore, the library would have to adapt to hardware differences such as different numbers of cores or different models of GPU. Even on instruction-set-compatible hardware such as different Intel processors, differences in cache geometry or memory architecture might require quite different decisions to achieve the highest performance.

In our approach, Elena instead designs an embedded domain-specific language (DSEL[1]) for Paul's problem domain. DSELs provide a concise, semantically well-founded way for a programmer to express a computation in a natural notation. The abstractions provided by a DSEL can be chosen to make compilation effective, e.g. through domain-specific parallelism constructs or by guiding the programmer to express intent rather than process. Elena uses our framework to embed the DSEL in Python and create a lightweight compiler that converts her DSEL into source code in an efficiency language such as C++ or CUDA, which exposes enough hardware properties to effect "last-mile" optimizations such as cache blocking and which is well supported by a rich ecosystem of optimizing compilers. Because our framework makes the DSEL compiler much more compact and easier to write than a full compiler (indeed, the DSEL compiler is written in Python and is typically hundreds of lines rather than thousands), most of Elena's effort is in expressing her strategy for mapping the computation to a specific hardware platform (or family of platforms) in the most efficient way. Her strategy may include producing a number of possible variants, each potentially optimal. The combination of procedural code and efficiency-language source code snippets that implement a DSEL compiler is called a *specializer* in our approach. The result is that although Paul's application is source- and performance-portable and expressed in a productivity language (Python in our case, though our approach generalizes to other languages), its performance is comparable to that of an equivalent application coded entirely in an efficiency-level language.

This report describes the following contributions:

1. The SEJITS methodology (Selective Embedded Just-in-Time Specialization) for embedding DSELs into a common productivity language and creating specializers that generate efficient code from them.
2. A framework called Asp that supports the embedding of DSELs in Python and implementation of lightweight DSEL compilers (specializers) in Python. Python is popular among productivity programmers and simplifies specializer creation for efficiency programmers: typical specializers are hundreds of lines long rather than thousands. Asp is implemented as a Python package that does not alter the interpreter in any way, ensuring existing Python code still runs.

[1] Following Hudak's [18] terminology, we use the acronym DSEL for Domain-Specific Embedded Languages to distinguish them from standalone or "external" DSLs.

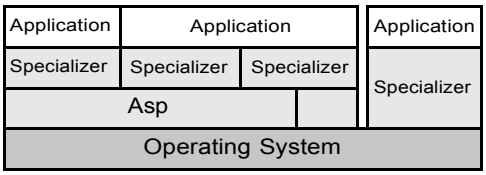


Figure 1: Applications interact with one or more specializers, which may use all, some, or none of the features provided by the Asp framework.

3. A demonstration of the approach via three implemented DSLs (specializers) for stencils[2],Gaussian mixture model training, and communication-avoiding sparse linear algebra [29]) that use our framework. Both the specializers and the nontrivial Python applications that use them perform comparably to manually-tuned efficiency-language code. One of the specializers (Gaussian mixture model) can target either a multicore CPU or different models of GPU transparently to the application programmer, and its performance is close enough to hand-tuned code that the research group whose speech-processing application relies on it has replaced their C++ implementation with a Python/specializer implementation to accelerate research.

4. A DSL that enables flexible filtering and customization of graph algorithms without sacrificing performance, by applying the SEJITS methodology to the Knowledge Discovery Toolkit (KDT), a framework for analyzing massive semantic graphs. We present a new roofline performance model [39] for high-performance graph computation, suitable for evaluating the performance of filtered semantic graph algorithms, and demonstrate that the strong scaling of our system for two different graph algorithms on graphs with millions of vertices and hundreds of millions of edges is very close to the best possible performance, despite allowing the algorithms to be expressed in the productivity language Python.

We do *not* in general claim that our approach improves on the performance of highly-tuned libraries (although in some cases it does), but rather that it allows the reuse of *strategies* for computing different kernels by enabling a degree of runtime tuning and auto-tuning that is often difficult to achieve gracefully with libraries, even while allowing application writers to express key parts of their algorithms in a high-level language while enjoying the performance of an efficiency language.

Section 3 describes the SEJITS methodology and what makes it different from other DSEL-based approaches. Section 4 describes the organization of typical specializers and the support provided by the Asp framework to simplify their creation. Section 5 describes four implemented specializers that use our framework and the performance of nontrivial applications that use them; all are either real customer applications maintained by researchers outside our group or standard domain benchmarks. That section also reflects on the results and discusses pros and cons of our approach. Section 6 compares our work to relevant prior work. Finally, Section 7 concludes with our thoughts on the role of the SEJITS approach for both productivity and efficiency programmers.

3 Methods, Assumptions and Procedures

3.1 The SEJITS Methodology

Selective Embedded Just-in-Time Specialization (SEJITS) describes a methodology for building DSEL compilers in high-level languages that bridge the gap between productive programming and high performance. In our approach the DSEL compiler includes domain-specific, imperative procedural code that captures domain-specific and hardware-specific knowledge, such as what code variants make sense to generate, how to tile or block memory, which partitioning strategy to use in subdividing the parallelism in the problem, and so on.

DSELs in other approaches usually transform constructs into lower-level constructs within the same language. This approach is common in languages that support metaprogramming, such as Scheme, Gambit, and Haskell. In contrast, our approach allows programmers to write specializers for languages without first-class metaprogramming, increasing the range of languages in which DSELs can be embedded. Furthermore, SEJITS concentrates on utilizing

[2]A stencil is a data-parallel structured grid computation in which each point in an N-dimensional grid is updated according to a function of its neighbor points. The function, boundary calculations, and the number of neighbors considered are application-specific.

the language's FFI (foreign function interface) for these specializers, allowing them to take advantage of downstream optimizing compilers by linking object code resulting from DSEL compilation with non-specialized productivity code in Python. This support for external compilers, combined with our framework's ability to easily "templatize" existing source code in efficiency languages such as C or CUDA, means that a specializer can start out as a simple "wrapping" of existing efficiency-level code (e.g. an efficiency programmer's specific solution to one problem instance) and gradually generalize the strategy to handle a wider range of problems, further simplifying the learning curve required to build a specializer.

A key advantage of our approach for specializer writers is that this procedural code can be written in a high-level language (in our case Python) and can leverage the extensive support we provide for "common" tasks such as abstract syntax tree manipulation, low-level code generation and caching, and so on. As a result, the typical specializer comprises a modest number of lines of Python code and does not require a deep understanding of compilers to create, because the source language is highly constrained and we provide a variety of building blocks for implementing specializers in a very high-level language.

In fact, the SEJITS approach can be thought of as providing two major capabilities: embedded DSLs that use the foreign function interface and run-time code generation with auto-tuning. These two pieces work together to enable all the benefits outlined above, but can in fact be used without each other. For example, even a library such as for sparse matrix multiplication, which can be thought of as "trivial" DSEL that expresses a single construct, can benefit from run-time code generation since the code can be tailored to the particular matrix and particular machine the computation is running on, yielding high performance. Such trivial DSELs are an important aspect of obtaining fast parallel performance from a productivity language. Thus, although the combination of FFI-enabled embedded DSELs and run-time code generation enables some of the most interesting uses of the approach, even high performance libraries can use the SEJITS approach.

3.2 Assumptions

We assume domain scientists will be able to express their applications using Python code, Python libraries, and Python-embedded DSELs as described in the foregoing sections. From the application writer's point of view, the DSELs behave like high-level libraries. We assume a POSIX-like environment in which standard build tools such as the *gcc* toolchain can be invoked at runtime; even environments such as Windows have this support via CygWin or similar packages.

We have created a number of specializers targeting different types of hardware, including Intel-compatible multicore CPUs and CUDA-capable GPUs (graphics processing units). Each specializer makes specific assumptions, noted in its description, about what hardware is available.

Our integration with the Knowledge Discovery Toolkit (KDT) assumes that KDT is already installed and running on the target hardware.

3.3 Procedures

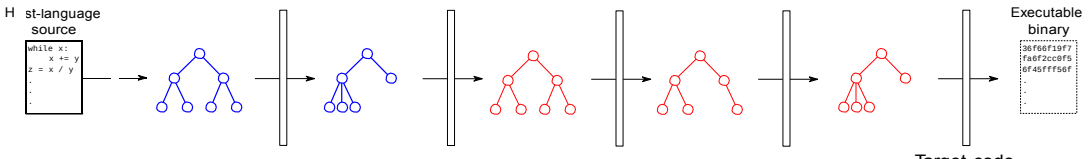
We implement and describe four specializers and some scientific applications that use them: stencils, Gaussian mixture model training, graph algorithms, and matrix powers. In our framework, each specializer is used by subclassing from a specific Python class and implementing specializer-specific virtual methods. Runtime translation is performed when these methods are subsequently called. Python code outside of any DSEL is executed normally by the Python interpreter; our approach does not modify the standard Python distribution in any way.

The individual subsections of Section 5 describe specific aspects of interest in each specializer's construction and show performance results of the specializer in a nontrivial application, thereby demonstrating the benefit to productivity programmers.

4 Specializers in the Asp Framework

Figure 1 summarizes how Asp[3], specializers, and applications interact in a system. Because specializers can be built even without the facilities provided by the Asp framework, only some of the specializers in the figure utilize it. Below we discuss features provided by the Asp framework and a prototypical architecture for specializer implementations.

[3]Asp is SEJITS for Python, http://github.com/shoaibkamil/asp



	Parser	Target-independent optimizations		Target-specific optimizations	Target code generation
Stencil specializer lines of code (§4.1)	101	67	159	84	0
KDT specializer lines of code (§4.3)	36	0	70	0	61

Figure 2: Lines of code in each phase of two specializers described in Section 5; each has some additional lines of utility code.

4.1 Framework features

The framework provides two main mechanisms for specializers to generate code: *templates* and *tree transformations*. Templates are efficiency-level code interspersed with productivity-code fragments and control code to fill in substrings based on properties determined at specialization time. These can embed values ranging in complexity from a single scalar value to complex efficiency-language statements, such as loop nests dependent on input properties for depth, and are rich enough that some specializers rely solely on templates.

For domains in which the efficiency-level code depends in a complex way on the DSEL source, we also provide support for tree transformations. It is common for specializers to use templates and tree transformations together to handle scenarios in which part of the output source code depends in a complex way on the input, while other parts such as helper functions are relatively static.

Since compilers are generally written as a multi-phase pipeline processing an intermediate tree data structure, Asp is built to facilitate rapid construction of specializers of this architecture in few lines of code.

The front-end parser is largely eliminated by the use of an embedded DSL, and back-end runtime code generation and caching from a target-language abstract syntax tree is supplied by Asp. Backends can target a variety of high-performance languages and compiler tools, leveraging the existing investment in these tools for target-specific optimization. We leverage CodePy (mathema.tician.de/software/codepy) for C/C++-based backends, and a Scala backend is in development.

For intermediate phases, our framework provides a visitor-pattern abstraction for tree traversal and a concise language for specifying strongly-typed intermediate representations. This intermediate representation reflects the semantics of the computation at hand, using the (imperative) Python source code as a declarative description of the computation. In other words, though computations in the DSELs are expressed as imperative code, the translation to the intermediate representation only translates *what* to compute, not how. The DSEL compiler is free to compute the declarative specification in whatever manner suits the domain and underlying hardware.

Other Asp features are targeted at specific phases. Each target language backend provides *default translations* for common DSEL constructs such as arithmetic and conditional expressions—in most cases, DSEL implementers only need to specify transformations for custom, domain-specific nodes.

During target-specific optimization, specializers rely on Asp to interrogate available hardware, to perform common generic optimizations such as loop unrolling and cache blocking, and to select the best among several generated code variants.

The combined effect of these features on code size reduction in specializers is dramatic, with complete production specializers such as the stencil and KDT specializers requiring only a few hundred lines of Python code, as shown in Figure 2.

Because specializers rely on generating source code and compiling it into dynamic shared libraries, caching is an important part of our approach as it amortizes the compilation time over many runs if subsequent calls can use the cached version. We leverage CodePy's integrated caching (which caches compiled code by comparing MD5 hashes of the source) but also include a higher-level caching method that allows specializers to dictate whether, based on runtime parameters, an existing version is runnable.

§	Specializer	Application	Logic	Tmpl.	Targets	Performance Remarks
5.1	Stencil (structured grid)	Bilateral image filtering	656	0	C++/OpenMP, Cilk+	91% of achievable peak based on roofline model [40]
5.2	Matrix powers ($\mathbf{A}^k\mathbf{x}$)	Conjugate gradient solver	200	2000	C/pthreads	2-4 times faster than SciPy
5.3	Gaussian mixture model (GMM) training	Speech diarization	800	3600	CUDA, Cilk+	CPU & GPU versions fast enough to replace original C++/pthreads code
5.4	Graph algorithms with KDT	Graph500 benchmark	325	0	C++/MPI	99% of performance of handcoding in C++

Figure 3: For each specializer we report the LOC of logic, LOC of templates, target languages, and a summary of the performance of the Python+SEJITS application compared to the original efficiency-language implementations. Recall that specializer logic is Python code that manipulates intermediate representations in preparation for code generation and templates are static efficiency- language "boilerplate" files into which generated code is interpolated. Our framework itself comprises 2094 LOC providing common functionality such as tree manipulation, code generation, compiler toolchain control, code caching, runtime hardware detection, and transforming common Python constructs such as simple arithmetic expressions into SM nodes.

4.2 Flexibility

Because they are invoked at runtime, specializers can generate different output code depending on the problem inputs (i.e. arguments to DSEL methods). A simple example is fixing the upper bound of a loop over an input matrix based on its runtime dimensions. A more sophisticated example occurs in the matrix powers specializer (Section 5.2), which samples statistics over matrix elements to help select the best blocking format for a particular problem.

Specialization may fail for a number of reasons: The code written by the application programmer may be valid in the host language, but not represent a valid DSEL program; the specializer may only be able to emit code for a compiler or hardware target that is unavailable at runtime; or downstream errors may be encountered in the code generation phase (e.g. invoking the compiler), due to errors in the specializer or misconfiguration. To this end, we recommend that each specializer include an implementation of its DSEL written purely in the embedding language, in our case Python. Asp automatically falls back to running this alternate version (albeit with poor performance) if specialization fails, improving source portability. For most specializers, a warning is issued in this case, to encourage programmers to express their code in the supported subset and explain the poor performance.

5 Results and Discussion

In this section we describe four implemented specializers, and for each one, report on its performance in the context of a widely-used scientific application. The specializers cover stencils (structured grid problems), Gaussian mixture model training, graph algorithms, and matrix powers. Each specializer subclasses from a specific Python class and implements specializer-specific virtual methods. Runtime translation is performed when these methods are subsequently called. Python code outside of any DSEL is executed normally by the Python interpreter; our approach does not modify the standard Python distribution in any way.

Figure 3 shows the four specializers on which we report, emphasizing the benefit of SEJITS to efficiency programmers by reporting on the approximate size of each specializer, a proxy for the efficiency programmers' effort to create it. The subsequent sections describe specific aspects of interest in each specializer's construction and show performance results of the specializer in a nontrivial application, thereby demonstrating the benefit to productivity programmers. Note that for each specializer, multiple applications have been developed using it; we outline only a single application due to space constraints.

For many of these domains, there is no universally acknowledged "gold standard" for comparing performance. We therefore compare against publicly-available, widely-used libraries and compilers for each domain that are generally accepted as providing good efficiency-level performance. Where possible, we also characterize performance as a portion of achievable peak based on hardware characteristics. For reported performance numbers, we elide JIT compilation time (which is on the order of seconds, mostly due to running an external compiler), since caching means that code generation and compilation occur only the first time a particular problem is run— subsequent executions occur at full speed.

```
class Div3D(StencilKernel):
  def kernel(self, in_grid1, in_grid2, in_grid3, out_grid):
    for x in out_grid.interior_points():
      for y in in_grid1.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C1*in_grid1[y]
      for y in in_grid2.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C2*in_grid2[y]
      for y in in_grid3.neighbors(x, 1):
        out_grid[x] = out_grid[x] + C3*in_grid3[y]
```

Figure 4: Python source code for 3D divergence kernel using the stencil DSEL. The user may specify grid connectivity or use defaults provided by the specializer. Note the inclusion of DSEL abstractions for `interior points` and neighbors, which avoids the analysis that optimizing compilers must perform when analyzing the ordered loops typical of an efficiency-language stencil implementation.

```
for (int x1x1=1; (x1x1<=256); x1x1=(x1x1+(1*256))) {
  for (int x2x2=1; (x2x2<=256); x2x2=(x2x2+(1*32))) {
#pragma omp parallel for
    for (int x1=x1x1; (x1<=min((x1x1+255),256)); x1=(x1+1)) {
      for (int x2=x2x2; (x2<=min((x2x2+31),256)); x2=(x2+1)) {
#pragma ivdep
        for (int x3=1; (x3<=(256-0)); x3=(x3+(4*1))) {
          //fully-unrolled neighbor loop,
          //unrolled further by 4
} } } } }
```

Figure 5: The presence of optimizations makes the optimized C++ source code the simple 3D divergence kernel harder to read and maintain. In Python, this is essentially two nested loops, while the optimized C++ is a 5-deep nest due to cache blocking, with loop bounds that are closely tied to the memory architecture of the target machine.

5.1 Stencils

Structured grid computations, also called stencil computations, consist of updating each grid point in an *n*-dimensional grid with some function of a subset of its neighbors. Stencils occur in image processing, solving linear equations, simulations of physical phenomena, and many other domains.

Why a Specializer: Depending on the specific application, the update function and the definition of a neighbor are highly problem-dependent. A general library would need to perform at least one function call per point. With more advanced techniques such as expression templates in C++, the operator can be inlined but optimizations cannot take advantage of many properties of the stencil. Although a stencil computation is conceptually straightforward, achieving good performance requires many optimizations, including blocking the computation in space and/or time explicitly [21, 11] or using cache-oblivious algorithms [16], vectorizing [30], or other techniques combining these optimizations such as polyhedral analysis [41].

The result is that the straightforward computation shown in Figure 4 becomes the complex and difficult-to-read code shown in Figure 5.

Details of Specializer: Our stencil specializer uses a cache-aware approach plus auto-tuning to generate fast, parallel efficiency code in either Intel Cilk+ or C++/OpenMP from the stencil DSEL shown in Figure 4. We have not yet implemented a GPU code generator, but nothing in the specializer structure precludes doing so.

The specializer implements two optimizations described in [21]: thread/cache blocking in phase 4 (target-class optimization) and register blocking in phase 5 (target-specific optimization). The class of stencils it can specialize is non-trivial but incomplete: only a single output grid is allowed, which precludes specializing many important kernels. Future work will expand this class, and any stencil outside the class is executed in pure Python with no changes to source code. All results in this paper use double-precision data, but the specializer also supports single-precision and will soon support integers.

Kernel Results: We show stencil specializer results for two kernels (a 3D 7-point Laplacian kernel and a 3D 7-point divergence kernel) which are discussed in detail in previous work on auto-tuning and occur in a variety of applications including climate simulation [21]. These computations are memory bandwidth-bound, and the roofline performance model [40] tells us the strategy to obtain the best performance is to reduce capacity cache misses.

Figure 6(a) compares the performance of the two kernels against that of the Pochoir stencil compiler [35], which offline-compiles a DSEL embedded in C++ into output targeted for the Intel C++ compiler. Unlike our cache-aware

algorithm, Pochoir implements a cache-oblivious algorithm and runs sub-problems using Intel Cilk+. Because Pochoir's cache-oblivious algorithm benefits from blocking across timesteps, we show results for both a single timestep and for the average over 5 timesteps of a stencil. In all cases, the pure-Python code took three to four orders of magnitude longer to run, and is therefore not shown.

For both the divergence and Laplacian kernels, the specializer slightly outperforms Pochoir for a single timestep. Since Pochoir can take advantage of temporal locality between timesteps (theoretically reducing the cache traffic to minimum [16]), we expect it to outperform our tuner for multiple timesteps; our results show that it is faster when taking advantage of temporal locality. We have very recently implemented DSEL extensions in the stencil specializer to express multiple timesteps, and modified the code generator to output explicit cache-aware code that reduces memory traffic to the theoretical minimum; this work is currently being prepared for publication. Therefore, we report only results from our non-timestep aware DSEL.

Performance as a percentage of single-timestep roofline peak is also in Figure 6. For the Laplacian kernel, our specializer obtains 91% of peak memory bandwidth, and 66% for the divergence kernel. Note that the peak memory bandwidth on the machine is dependent on the number of memory streams; thus the peak for the divergence kernel is higher than for the Laplacian kernel.

Application Results: The application is a 3D bilateral filter for reducing noise and enhancing important features in MRI (magnetic resonance imaging) images of the brain [28]. It combines spatial and signal weights to create an edge-preserving smoothing filter. Such an application requires applying the filter with varying radii to highlight features of different sizes. The application loads MRI data using the NiBabel neuro-imaging library (nipy.sourceforge.net) and displays output data using matplotlib (matplotlib.sourceforge.net), both conveniently available to Python programmers; this illustrates the benefit of selecting a popular common embedding language with broad library support.

Unlike the previous two kernels, the bilateral filter uses all neighboring points within the set radii; for example, this means it is a 27-point stencil with $r = 1$ and a 343-point stencil with $r = 3$. In addition, the weight of each neighbor point is determined through an indirect table lookup into a small array, indexed by the intensity difference and multiplied by a function of the distance; this lookup can potentially result in much slower performance. For this application, the roofline model shows it is bound by in-core computation (because the lookup table is in cache, the computation required to compute the index is more costly), and from the mix of floating point and non-floating point operations, we can expect to obtain at most 62% of peak floating point performance on our test machine.

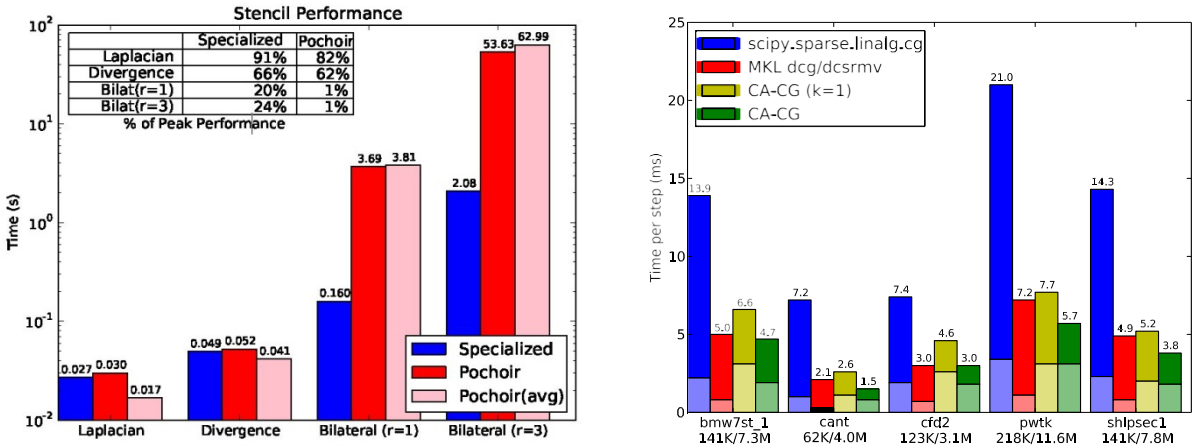
Figure 6(a) shows the performance of the computationally-intense portion of the bilateral filter application. Remarkably, the performance is even better than what is obtained by Pochoir, perhaps due to assumptions the Pochoir compiler makes about the stencil function (particularly related to indirect accesses). As shown in Figure 6(a), the specialized bilateral filter kernel obtains up to 24% of maximum floating point performance, which could be increased were our code better vectorizable. This performance is over three orders of magnitude faster than pure Python and is excellent for a computation-bound problem.

It is important to contextualize these performance gains by also examining the productivity improvement realized by using our stencil DSEL. As a gross metric, the lines of code for the optimized generated stencils are at least an order of magnitude larger; in addition, they encapsulate a large amount of domain-specific tuning knowledge, such as which order to traverse the grids, blocking for cache and registers/vectorization, etc. Furthermore, the optimal parameters change depending on the particular stencil problem. Pochoir helps reduce the necessity of programmers needing to know these low-level details to some extent, but is hampered by the need to write code in C++ as well as requiring an offline-compilation process. Overall, the stencil specializer produces very fast code with high productivity, enabling domain programmers to write in a high-level language and gain the performance of hand-tuned parallel low-level programming.

5.2 Communication-Avoiding Sparse Linear Algebra

Many algorithms for solving sparse linear systems ($Ax = b$), or for finding eigenvalues of a sparse matrix, are iterative processes that access the matrix A with one or more sparse matrix-vector multiplications (SpMVs) per iteration. Since an SpMV must read a matrix entry from memory for every 2 useful floating-point operations, Demmel et al. have proposed *communication-avoiding* algorithms that improve performance by trading redundant computation for memory traffic [29]. Both serial and parallel implementations can benefit from these algorithms, as communication in the serial case refers to memory-to-cache traffic.

We have implemented a specializer for a communication-avoiding *matrix powers kernel* (so-called $\mathbf{A}^k\mathbf{x}$). Matrix powers computes $Ax, A^2x, \ldots, A^kx$ (or some equivalent basis that spans the same vector space) for matrix A, vector



(a) Stencil (b) CG solver using Akx

Figure 6: (a) Summary of stencil performance (log-scale) on an Intel Core i7 870 (2.93 GHz) for 3D grid sizes of 258^3. Bilateral filter radius is shown as *r*. Since Pochoir can benefit from temporal locality across timesteps, we also show the average per-timestep time for Pochoir over 5 iterations. In all cases, the pure Python performance (not shown) was at least 3 orders of magnitude slower than specialized performance. (b) Conjugate Gradient solver performance using communication-avoiding matrix powers kernel on a dual-socket Intel Xeon X5550 (2.67 GHz) on test matrices from finite-element and fluid dynamics applications (www.cise.ufl.edu/research/sparse/matrices). A matrix labeled 141K/7.3M has 141K rows and 7.3M nonzero elements. The dark part of each bar shows time spent on matrix powers while the light part shows time in the remainder of the solver. Note that the convergence properties are at most 4.5% worse for 10^{-6} reduction in $|r|^2$; however, we report time per step since the decision to use CA-CG is independent of our tuner.

x, and a small constant *k*. $\mathbf{A}^k\mathbf{x}$ is an important ingredient in Krylov-subspace solvers such as Conjugate Gradient (CG), because once the computation has been performed, the next *k* steps of the solver can proceed without further memory accesses to *A* by combining vectors from this set.

Why a Specializer: Although this specializer does not lower any user-provided code, the tuning logic associated with blocking and tiling (which requires inspecting input values, as described below) was easy to write in Python. To the best of our knowledge, our $\mathbf{A}^k\mathbf{x}$ specializer is the first publically available productivity-friendly implementation of communication-avoiding $\mathbf{A}^k\mathbf{x}$.

Details of Specializer: The specializer partitions the set of matrix rows into cache blocks, and computes a cache block's entries in all *k* output vectors before moving on to the next cache block. This will only work if the matrix structure is such that the dependencies between cache blocks do not get too large. In addition to cache-blocking the matrix, the usual SpMV optimization of register tiling reduces the memory size of the matrix and makes it possible to use SIMD instructions. The specializer generates code for different blocking and tiling formats using templates, and chooses among them by auto-tuning. (The choice of *k* must be made outside the specializer, as it affects the rest of a Krylov-subspace method too.)

Kernel and Application Results: We have implemented a Communication-Avoiding Conjugate Gradient (CG) solver, the simplest useful Krylov-subspace method solver, in Python. The CG solver uses the $\mathbf{A}^k\mathbf{x}$ specializer and calls the Intel Math Kernel for the operations other than matrix powers. Although MKL is well-optimized, the ability to compose the $\mathbf{A}^k\mathbf{x}$ computation with the solver's subsequent dot product operations would avoid having to read the vectors from memory an extra time. (As described later, specializer composition is an important avenue for future work.) Nonetheless, our CG solver runs several times faster than SciPy's serial and non-communication-avoiding efficiency language implementation, and in fact is even faster than MKL's optimized parallel CG implementation. The convergence properties for our test matrices differ slightly between the two CG algorithms on a few of the matrices (at worse requiring 4.5% more multiplies), but the decision as to whether the difference is meaningful depends on the application.

Figure 6(b) shows the results, demonstrating both the algorithmic and auto-tuning benefits of our CG implementation.

5.3 Gaussian Mixture Model Training

Gaussian Mixture Models (GMMs) are a class of statistical models used in speech recognition, image segmentation, document classification and numerous other areas. To apply GMM-based techniques to a particular problem, GMMs must be "trained" to match a set of observed data. The most common training algorithm, Expectation Maximization (EM), is computationally intensive, iterative, and highly data-parallel, making it particularly amenable to specialization for multicore hardware with wide SIMD vector support.

Why a Specializer: Current parallel implementations of the EM algorithm such as [32] employ a fixed strategy for mapping the algorithm's data-parallelism onto the parallel hardware. However, the best-performing mapping for various EM algorithm substeps depends on both the GMM problem size and hardware platform parameters [10].

Details of Specializer: The specializer can emit either CUDA or Cilk+ code using two included sets of templated implementations. The specializer selects the best algorithm variant at runtime based on the size of the training problem and the available hardware, relying primarily on the SEJITS framework's templating mechanisms to instantiate the variant.

Kernel Results: Figure 7(a) shows that the GMM training performance of our specializer can beat even the handcoded CUDA [32] implementation by selecting the best-performing algorithmic variant at runtime based on training problem size [10]. The specializer can emit CUDA and Cilk+ code, making it performance-portable both within and across architecture families with no changes to client Python applications like the one we describe below.

We are confident that future work on specializers for other components of the application will allow us to meet the domain target of 200×RT (see next section).

Application Results: Our target application, a meeting diarizer [1] that identifies the number of speakers in a recorded meeting and determines who spoke when, originally consisted of about 3000 lines of C++ with pthreads. Repeated GMM training was the performance bottleneck. The new implementation consists of about 100 lines of Python implementing the body of the application, plus a specializer consisting of about 800 lines of Python and 3600 lines of CUDA and Cilk+ templates. The reason the specializer is so large is because it integrates a large number of tuned implementations that were not present in the original application's GMM implementation, including support for multiple backends. Note also that the specializer is not specific to this application; that is, a number of other applications have been developed that utilize the specializer, in a variety of domains [10].

Speech recognition domain experts evaluate performance according to the real-time factor (×RT) metric. For example, 100×RT means that 1 second of audio can be processed in 1/100 second. An important domain target is ≥ 200×RT, at which point online approaches to speaker diarization become fast enough to obviate further research in offline approaches. (We have not yet investigated what components of the processing pipeline should be specialized next in order to make further progress towards that goal.)

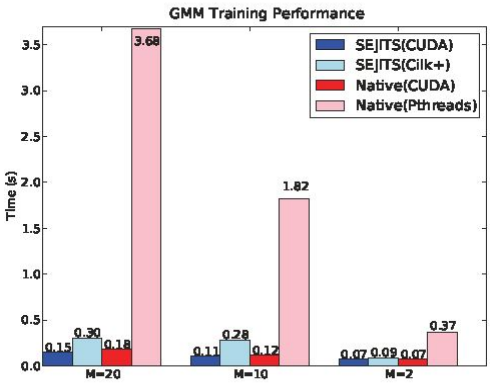
Figure 7(b) shows that our Python implementation with specialized EM training achieves 50% of that goal, using the CUDA and Cilk+ specializers, while the original C++/pthreads application only gets to 10%.

5.4 Integrating SEJITS with KDT for Graph Algorithms

Large-scale graph analytics are a central requirement of bioinformatics, finance, social network analysis, national security, and many other fields. Going beyond simple searches, analysts use high-performance computing systems to execute complex graph algorithms on large corpora of data. Often, a large semantic graph is built up over time, with the graph vertices representing entities of interest and the edges representing relationships of various kinds—for example, social network connections, financial transactions, or interpersonal contacts.

In a semantic graph, edges and/or vertices are labeled with *attributes* that may represent (for example) a timestamp, a type of relationship, or a mode of communication. An analyst (i.e. a user of graph analytics) may want to run a complex workflow over a large graph, but wish to only use those graph edges whose attributes pass a filter defined by the analyst. For example, consider a graph whose vertices are Twitter users, and whose edges represent two different types of relationships between users. In the first type, one user "follows" another; in the second type, one user "retweets" another user's tweet. Each retweet edge carries as attributes a timestamp and a count. An example query we will use in this section is a breadth-first search (BFS) of vertices reachable from a particular user via the subgraph consisting only of "retweet" edges with timestamps earlier than June 30, 2009.

The Knowledge Discovery Toolkit [26] is a flexible Python-based open-source toolkit for implementing complex semantic graph algorithms and executing them on high-performance parallel computers. KDT achieves high performance by invoking computational primitives supplied by a parallel C++/MPI backend, the Combinatorial BLAS or



(a) GMM training

Mic Array	Orig. C++/pthreads	Py+Cilk+	Py+CUDA
	Westmere	Westmere	GTX285/GTX480
Far field	11×	32×	68×/71×

(b) Diarizer performance

Figure 7: (a) Performance of GMM training over a range of number of components in the mixture model (M), which in many applications varies as the application algorithm converges. CUDA results are from an NVIDIA GTX480. Cilk+ results are from dual-socket Intel X5680 Westmere (3.33 GHz). "Pangborn" is the original native CUDA implementation from [32]. (b) Diarizer application performance as a multiple of real time; "100×" means that 1 second of audio can be processed in 1/100 second. The Python application using CUDA and Cilk+ outperforms the native C++/pthreads implementation by a factor of 3-6.

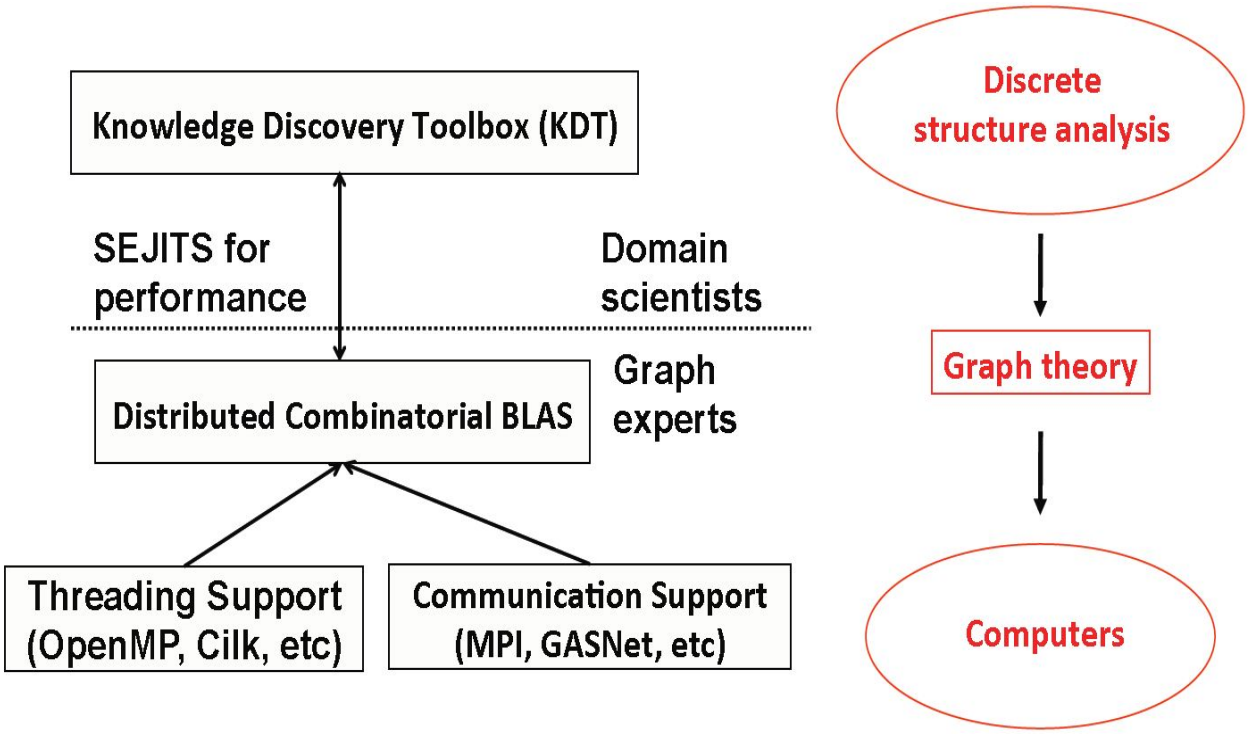


Figure 8: Overview of the high-performance graph-analysis software architecture described in this section. KDT has graph abstractions and uses a very high-level language. Combinatorial BLAS has sparse linear-algebra abstractions, and geared towards performance.

CombBLAS [7]. To exploit this performance, an expert user must be able to express a desired graph computation (graph traversal and filtering) as sparse matrix computations in terms of algebraic semiring operations, such as the tropical (min,+) semiring for shortest paths, or the real (+,*) semiring/field for numerical computation. Two fundamental kernels in CombBLAS, sparse matrix-vector multiplication (SpMV) and sparse matrix-matrix multiplication (SpGEMM), then explore the graph by expanding existing frontier(s) by a single hop. The semiring scalar multiply operation determines how the data on a sequence of edges are combined to represent a path, with a filtered multiply returning a "null" object (formally, the semiring's additive identity or SAID) if the filter predicate is not satisfied. The semiring scalar add operation determines how to combine two or more parallel paths. For example, Figure 9 (top) shows the scalar multiply operation for our running example of BFS on a Twitter graph. The usual semiring multiply for BFS is select2nd, which returns the second value it is passed; the multiply operation returns the second value if the filter succeeds.

Filters expressed this way have high performance because the number of calls to the filter operations is asymptotically the same as the minimum number of necessary calls to the semiring scalar multiply operation, and the filter itself is a local operation that uses only the data on one edge. However, translating a graph computation into its equivalent semiring algebra representation is beyond the expertise of most users interested in graph analytics. Therefore, recent versions of KDT also allow users to instead define graph filters as predicates that act to modify KDT's action based on the attributes that label individual edges or vertices. Users define these filters in Python, but doing so results in a serialized upcall to Python on every operation, slowing performance by nearly two orders of magnitude—80×—compared to defining the filters directly in C++or expressing the computation in semiring linear algebra. This section presents new work that allows nonexpert KDT users to define Python filters *without paying the performance penalty of Python upcalls*.

To accomplish this, we use SEJITS to define two semantic-graph-specific domain-specific languages (DSL), as shown in Figure 11(a,b): one for filters and one for the user-defined scalar semiring operations for flexibly implementing custom graph algorithms. These implement the specialization necessary for filters and semirings written in (a subset of) Python to execute as efficiently as low-level C++ code. The DSLs are proper subsets of Python with normal Python syntax, but they restrict the kinds of operations and constructs that users can utilize in filters and semiring operations. The filter DSEL expresses what a filter can do: a filter takes in one or two inputs (that are of pre-defined edge/vertex types), must return a Boolean, and is allowed to do comparisons, accesses, and arithmetic on immediate values and edge/filter instance variables. In addition, to facilitate translation, we require that a filter be an object that inherits from the PcbFilter Python class, and that the filter function itself use Python's usual interface for callable objects, requiring the class define a function __call__ . Figure 9 shows an "old style" KDT filter in native C++, its expression in our filter DSEL, and how it would be added to a graph in SEJITS-enhanced KDT. The filter DSEL version defines a fully-valid Python class that can be translated into C++ since it only uses constructs that are part of our restricted subset of Python. For the semiring DSEL, unary and binary operations used in semirings and other operations in KDT are similarly defined, but must inherit from the PcbFunction class and must return one of the inputs or a numeric value that corresponds to the KDT built-in numeric type.

As with our other DSELs, the first time a specialized filter or semiring function is called from Python, its source code is introspected (using existing Python facilities) to get the Abstract Syntax Tree (AST) in Python. The appropriate specializer (filter or semiring) maps this AST to a domain-specific AST for that DSEL; each specializer defines tree transformations that dictate how this occurs. For example, the Python function definition for __call__ is translated into either a UnaryPredicate or BinaryPredicate node in the case of the filter embedded DSL, depending on how many inputs it accepts. The resulting domain-specific AST must comply with the filter or semiring *semantic model*, which defines the semantics of valid translatable objects. The semantic models for both filter and semiring DSELs allow the user to examine the input data types, do comparisons, and perform arithmetic on fields. Figure 10 shows the semantic model for the filter DSEL. ASTs conforming to this semantic model are "correct-by-construction", that is, they obey the restrictions of what can be safely translated. For example, we require that the return value of a filter be provably a Boolean (by forcing the BoolReturn node to have a Boolean body), and that there is either a single input or two inputs (either UnaryPredicate or BinaryPredicate). If a user-provided filter or semiring operation violates the corresponding semantic model, we run it in pure Python as usual: unspecializable code still runs correctly, albeit much more slowly.

After translation into an AST compliant with the appropriate semantic model, the rest of the translation is straightforward, utilizing Asp's infrastructure for converting semantic models into backend code. For many of these transformations, defaults built into Asp are sufficient; for example, we leverage the default translation for Python arithmetic operations and numeric constants. The end result of this step is source code in C or C++ containing the function in a private namespace plus some KDT glue code. This source file is passed to CodePy, which compiles it into a small dynamic link library that is then automatically loaded into the running Python interpreter. Finally, because of code caching built into CodePy, all calls after the first will just directly call the specialized and compiled function, skipping

```
ParentType multiply( const TwitterEdge & arg1,
                     const ParentType & arg2)
{
    time_t end = stringtotime("2009/06/30");
    if (arg1.isRetweet() && arg1.latest(end))
        return arg2; // unfiltered multiply yields normal value
    else
        return ParentType(); // filtered multiply yields SAID
}
```

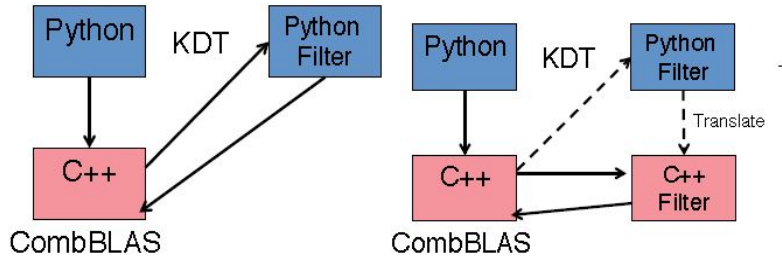
```
class MyFilter(PcbFilter):
  def __init__(self, ts):
    self.ts = ts
  def __call__(self, e):
    # if it is a retweet edge
    if (e.isRetweet and
        # and it is before our initialized timestamp
        e.latest < self.ts):
      return True
    else:
      return False
```

```
# G is a kdt.DiGraph
def earlyRetweetsOnly(e):
  return e.isRetweet() and e.latest < str_to_date("2009/06/30")

G.addEFilter(earlyRetweetsOnly)
G.e.m__aterializeFilter() # omit this line for on-the-fly filtering

# perform some operations or queries on G

G.delEFilter(earlyRetweetsOnly)
```

Figure 9: Top: An example of a filtered scalar semiring operation in Combinatorial BLAS. This multiply operation only traverses edges that represent a retweet before June 30, 2009. Middle: Example of an edge filter that the translation system can convert from Python into fast C++ code. Note that the timestamp in question is passed in at filter instantiation time. Bottom: Adding and removing an edge filter in KDT, with or without materialization.

```
UnaryPredicate(input=Identifier, body=BoolExpr)
BinaryPredicate(inputs=Identifier*, body=BoolExpr)
    check assert len(self.inputs)==2
Expr = Constant | Identifier | BinaryOp | BoolExpr
Identifier(name=types.StringType)
BoolExpr = BoolConstant | IfExp | Attribute | BoolReturn |
           Compare | BoolOp
Compare(left=Expr, op=(ast.Eq | ast.NotEq | ast.Lt |
        ast.LtE | ast.Gt | ast.GtE), right=Expr)
BoolOp(op=(ast.And | ast.Or | ast.Not), operands=BoolExpr*)
    check assert len(self.operands)<=2
Constant(value = types.IntType | types.FloatType)
BinaryOp(left=Expr, op=(ast.Add | ast.Sub), right=Expr)
BoolConstant(value = types.BooleanType)
IfExp(test=BoolExpr, body=BoolExpr, orelse=BoolExpr)
# this is for a.b
Attribute(value=Identifier, attr=Identifier)
BoolReturn(value = BoolExpr)
```

Figure 10: Semantic Model for KDT filters using SEJITS. The semantic model for semiring operations is similar, but instead of enforcing boolean return values, enforces that the returned data item be of one of the input return types.



(a) KDT without SEJITS (b) KDT with SEJITS

	First Run	Subsequent
Codegen	0.0545 s	0 s
Compile	4.21 s	0 s
Import	0.032 s	0.032 s

(c) SEJITS overhead

Figure 11: (a) Calling process for filter operations in KDT (semiring operations is similar). For each edge, the C++ infrastructure must upcall into Python to execute the callback. (b) Using our DSLs, the C++ infrastructure calls the translated version of the operation, eliminating the upcall overhead. (c) Overheads of using the filtering DSL (on a 36-core machine). After the first call, subsequent calls only incur the penalty of Python's import statement, which loads the cached library.

the specialization overhead. While this overhead does not exist when using native CombBLAS, it is trivial compared to the penalty of upcalling into Python, as Figure 11(c) shows.

To integrate this translation machinery with KDT, we modified the normal KDT C++ filter objects, which are instantiated with pointers to Python functions, by adding a function pointer that is checked before executing the upcall to Python. This function pointer is set by the SEJITS translation machinery to point to the translated function in C++. We similarly modify KDT's C++ function objects used for binary and unary operations. For both kinds of objects, the functions or filters are type-specialized using user-provided information. Future refinements will allow inferred type-specialization.

We evaluate our approach using two graph queries, running each on both synthetic and real data. Our first query is a filtered graph traversal: Given a vertex of interest, determine the number of hops required to reach each other vertex by using only retweeting edges timestamped earlier than a given date. The query is therefore a breadth-first search (BFS) in the graph that ignores edges that do not pass the filter. We generate synthetic data using the R-MAT [24] model, which can generate graphs with a very skewed degree distribution. An R-MAT graph of scale N has 2^N vertices and approximately *edgefactor* $\cdot 2^N$ edges. In our tests, our *edgefactor* is 16, and our R-MAT seed parameters a, b, c, and d are 0.59, 0.19, 0.19, 0.05 respectively. After generating this non-semantic (Boolean) graph, the edge payloads are artificially introduced using a random number generator in a way that allows us to set the target filter's permeability (fraction of edges that will pass the filter). We also evaluate the query on anonymized Twitter data [23, 42] in which graph edges can represent two different types of interactions: one type of edge encodes the "following" relationship (a

Table 1: Sizes (vertex and edge counts) of different combined Twitter graphs.

Label	Vertices (millions)	Edges (millions)		
		Tweet	Follow	Tweet&follow
Small	0.5	0.7	65.3	0.3
Medium	4.2	14.2	386.5	4.8
Large	11.3	59.7	589.1	12.5
Huge	16.8	102.4	634.2	15.6

Table 2: Statistics about the largest strongly connected components of the Twitter graphs

	Vertices	Edges traversed	Edges processed
Small	78,397	147,873	29.4 million
Medium	55,872	93,601	54.1 million
Large	45,291	73,031	59.7 million
Huge	43,027	68,751	60.2 million

directed edge from v_i to v_j means that v_i is following v_j) and another type encodes an abbreviated "retweet" relationship (a directed edge from v_i to v_j means that v_i has mentioned v_j at least once in his tweets). Retweet edges also track the number of such tweets (count) as well as the last tweet date if count is larger than one.

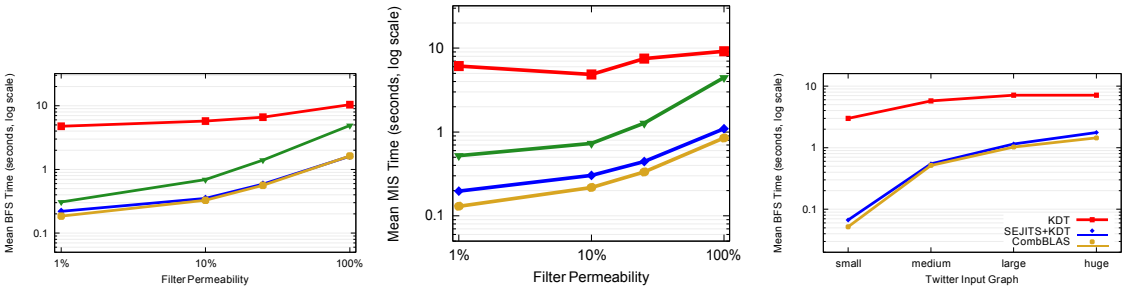
Our second query is to find the maximal independent set (MIS) of this graph. MIS finds a subset of vertices such that no two members of the subset are connected to each other and all other vertices outside MIS are connected to at least one member of the MIS. Since MIS is defined on an undirected graph, we first ignore edge directions, then we execute Luby's randomized parallel algorithm [25] implemented in KDT. The filter is the same as in the first query. We run this query on Erdős-Rényi graphs [14] with an *edgefactor* of 4 because MIS on R-MAT graphs complete in very few steps due to high coupling, barring us from performing meaningful performance analysis.

More details for these four different (small-huge) combined graphs is listed in Table 1. Contrary to the synthetic data, the real Twitter data is directed and we only report BFS runs that hit the largest strongly connected component of the filter-induced graphs. More information on the statistics of the largest strongly connected components of the graphs can be found in Table 2. Processed edge count includes both the edges that pass the filter and the edges that are filtered-out.

The tweets occurred in the period of June-December of 2009. To allow scaling studies, we created subsets of these tweets, based on the date they occur. The *small* dataset contains tweets from the first two weeks of June, the *medium* dataset contains tweets that happened in June and July, the *large* dataset contains tweets dated June-September, and finally the *huge* dataset contains all the tweets from June to December. These partial tweets are then induced upon the graph that represents the follower/followee relationship. If a person tweeted someone or has been tweeted by someone, then the vertex is retained in the tweet-induced combined graph.

We examine graph analysis behavior on two platforms. Mirasol is a single-node platform composed of four Intel Xeon E7-8870 processors. Each socket has ten cores running at 2.4 GHz, and supports two-way simultaneous multithreading (20 thread contexts per socket). The cores are connected to a very large 30 MB L3 cache via a ring architecture. The sustained stream bandwidth is about 30 GB/s per socket. The machine has 256 GB 1067 MHz DDR3 RAM. We use OpenMPI 1.4.3 with GCC C++ compiler version 4.4.5, and Python 2.6.6. Hopper is a Cray XE6 massively parallel processing (MPP) system, built from dual-socket 12-core "Magny-Cours" Opteron compute nodes. In reality, each socket (multichip module) has two dual hex-core chips, and so a node can be viewed as a four-chip compute configuration with strong NUMA properties. Each Opteron chip contains six super-scalar, out-of-order cores capable of completing one (dual-slot) SIMD add and one SIMD multiply per cycle. Additionally, each core has private 64 KB L1 and 512 KB low-latency L2 caches. The six cores on a chip share a 6MB L3 cache and dual DDR3-1333 memory controllers capable of providing an average STREAM[27] bandwidth of 12 GB/s per chip. Each pair of compute nodes shares one Gemini network chip, which collectively form a 3D torus. We use Cray's MPI implementation, which is based on MPICH2, and compile our code with GCC C++ compiler version 4.6.2 and Python 2.7. Complicating our experiments, some compute nodes do not contain a compiler, which is necessary for SEJITS to perform runtime specialization; we ensured that a compute node with compilers available was used to build the SEJITS+KDT filters.

Figure 12(a) shows the relative distributed-memory performance of four methods in performing breadth-first search



(a) BFS Synthetic (Hopper) (b) BFS Synthetic (Mirasol) (c) BFS Twitter (Mirasol)

Figure 12: (a) Relative breadth-first search performance of four methods: native C++ CombBLAS (yellow), SEJITS semiring DSL (blue), KDT with SEJITS-specialized Python filters (green), and KDT with unspecialized Python filters (red), using 24 Hopper nodes each containing two 12-core AMD processors. The y-axis is in seconds on a log scale. (b) Relative maximal independent set performance of the same four methods using 26 cores of Intel Xeon E7-8870 processors. (c) Relative filtered-breadth-first-search performance on real Twitter data using KDT (pure Python), KDT+SEJITS (runtime specialization of Python), and native CombBLAS (hand-coded C++), using 16 cores of Intel Xeon E7-8870 processors.

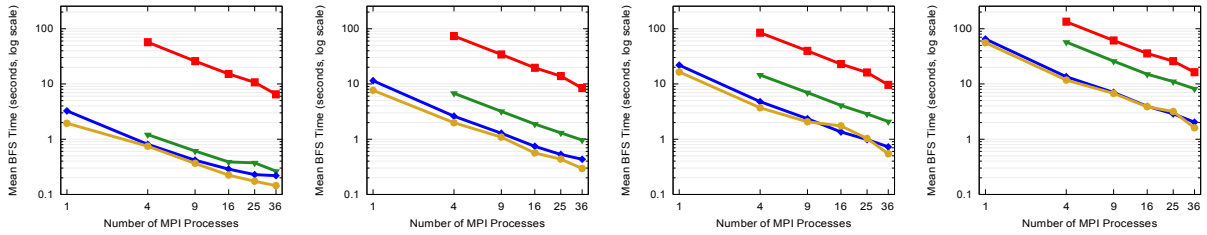
on a graph with 32 million vertices and 512 million edges, with varying filter permeability, on Hopper. The structure of the input graph is an R-MAT of scale 25, and the edges are artificially introduced so that the specified percentage of edges pass the filter. These experiments are run on Hopper using 576 MPI processes with one MPI process per core. The SEJITS/SEJITS KDT implementation closely tracks CombBLAS performance, with the gap between it (blue line) and the Python/SEJITS KDT implementation (green line) growing as permeability increases. This is expected because as the permeability increases, more semiring operations are performed, making Python based semiring operations a bottleneck. We also include the performance of a materialized filter, a different filtering strategy in which the filter is first applied to the entire graph resulting in a new copy of the graph with the filter applied. KDT supports this capability, but we did not augment it with SEJITS; however, we report its performance here because in this example the filter rate is so high that it showcases the situation in which the filter restricts the query to a localized neighborhood. In such cases, the on-the-fly filter would avoid touching most of the graph while the materialized filter needs to process all the edges. The effects of filter permeability on the MIS performance are shown in Figure 12(b).

Figure 12(c) shows the relative performance of four systems in performing breadth-first search on real graphs that represent the Twitter interaction data on Mirasol. We chose to present 16 core results because that is the concurrency in which this application performs best, beyond which synchronization costs start to dominate due to the large diameter of the graph after the filter is applied. Since filter to semiring operations ratio is very high (on the order of 200-1000), SEJITS translation of the semiring operation did not change the running time. Therefore, we only include a single SEJITS+KDT line to avoid cluttering the plot. SEJIT+KDT's performance is identical to the performance of CombBLAS in these data sets, showing that for real-life inspired cases, our approach is as fast as the underlying high-performance library.

Since SEJITS specializes both the filter and the semiring operation, we discuss the effects of each specialization separately. CombBLAS achieves remarkable linear scaling with increasing process counts (34-36X on 36 cores), while SEJITS+KDT closely tracks its performance and scaling: All of the performance plots show that the performance when both the filter and the semiring are specialized with SEJITS is very close to the CombBLAS performance. (Single core KDT runs did not finish in a reasonable time to report.) We do not report performance of materialized filters as they were previously shown to be the slowest.

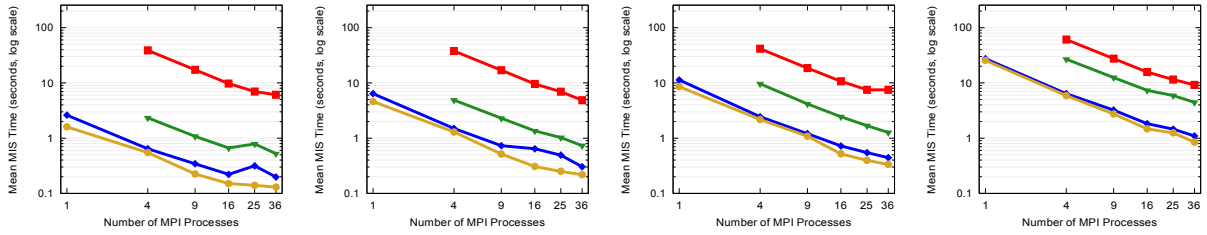
The Python/SEJITS case is typically slower than the SEJITS/SEJITS case, with the gap depending on the permeability as discussed previously. In the BFS case, shown in Figure 13, Python/SEJITS is $3-4\times$ slower than SEJITS/SEJITS when permeability is 100% due to the high number of semiring operations, but only $20-30\%$ slower when permeability is 1%. The scaling of MIS, shown in Figure 14, is more sensitive to semiring translation, even for low permeabilities. The semiring operation in the MIS application is more computationally intensive, hence specializing semirings become more important in MIS.

Parallel scaling studies of BFS at higher concurrencies is run on Hopper, using the scale 25 synthetic R-MAT data set. Figure 15 shows the comparative performance of KDT on-the-fly filters, SEJITS+KDT specializing filters



(a) 1% permeable (b) 10% permeable (c) 25% permeable (d) 100% permeable

Figure 13: Parallel 'strong scaling' results of filtered BFS on Mirasol, with varying filter permeability on a synthetic data set (R-MAT scale 22). Both axes are in log-scale, time is in seconds.



(a) 1% permeable (b) 10% permeable (c) 25% permeable (d) 100% permeable

Figure 14: Parallel 'strong scaling' results of filtered MIS on Mirasol, with varying filter permeability on a synthetic data set (Erdős-Rényi scale 22). Both axes are in log-scale, time is in seconds.

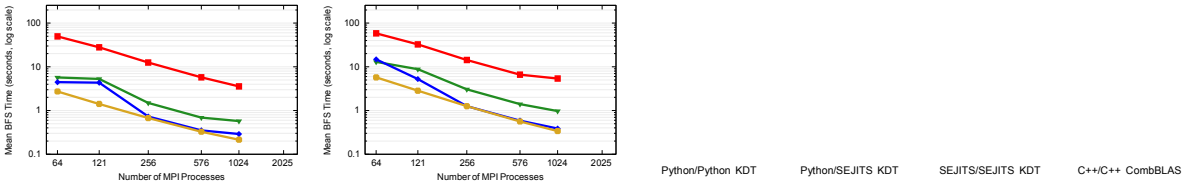
only, SEJITS+KDT specializing both filters and semiring operations, and native CombBLAS, with 10% and 25% filter permeability.

5.5 A roofline model of Breadth-first Search

In addition to evaluating the performance improvement of KDT when using SEJITS rather than pure Python for filters and semiring operations, we also show that our performance compares favorably to the *best performance achievable in principle* on breadth-first search (BFS) graph algorithms using SEJITS-enhanced KDT. In this section, we extend the Roofline model [39] to quantify the performance bounds of BFS as a function of optimization and filter success rate. The Roofline model is a visually intuitive representation of the performance characteristics of a kernel on a specific machine. It uses bound and bottleneck analysis to delineate performance bounds arising from bandwidth or compute limits. In the past, the Roofline model has primarily been used for kernels found in high-performance computing. These kernels tend to express performance in floating-point operations per second and are typically bound by the product of arithmetic intensity (flops per byte) and STREAM [27] (long unit-stride) bandwidth. In the context of graph analytics, none of these assumptions hold.

In order to model BFS performance, we decouple in-core compute limits (filter and semiring performance as measured in processed edges per second) from memory access performance. The in-core filter performance limits were derived by extracting the relevant CombBLAS, KDT, and SEJITS+KDT versions of the kernels and targeting arrays that fit in each core's cache. We run the edge processing inner kernels 10000 times (as opposed to once) to obfuscate any memory system related effects to get the in-core compute limits.

Analogous to arithmetic intensity, we can quantify the average number of bytes we must transfer from DRAM per edge we process — bytes per processed edge. In the following analysis, the indices are 8 bytes and the edge payload is 16 bytes. BFS exhibits three memory access patterns. First, there is a unit-stride *streaming* access pattern arising from access of vertex pointers (this is amortized by degree) as well as the creation of a sparse output vector that acts as the new frontier (index, parent's index). The latter incurs 32 bytes of traffic per traversed edge in write-allocate caches assuming the edge was not filtered. Second, access to the adjacency list follows a *stanza-like* memory access pattern. That is, small blocks (stanzas) of consecutive elements are fetched from effectively random locations in memory. These stanzas are typically less than the average degree. This corresponds to approximately 24 bytes (16 for payload and 8 for index) of DRAM traffic per processed edge. Finally, updates to the list of visited vertices and the indirections when accessing the graph data structure exhibit a memory access pattern in which effectively *random* 8 byte elements are



(a) 10% permeable (b) 25% permeable (c) Legend

Figure 15: Parallel 'strong scaling' results of filtered BFS on Hopper, with varying filter permeability on a synthetic data set (R-MAT scale 25). Both axes are in log-scale, time is in seconds.

Table 3: Statistics about the filtered BFS runs on the R-MAT graph of Scale 23 (M: million)

Filter permeability	Vertices visited	Edges traversed	Edges processed
1%	655,904	2.5 M	213 M
10%	2,204,599	25.8 M	250 M
25%	3,102,515	64.6 M	255 M
100%	4,607,907	258 M	258 M

updated (assuming the edge was not filtered). Similarly, each visited vertex generates 24 bytes of random access traffic to follow indirections on the graph structure before being able to access its edges. In order to quantify these bandwidths, we wrote a custom version of STREAM that provides stanza-like memory access patterns (read or update) with spatial locality varying from 8 bytes (random access) to the size of the array (STREAM).

The memory bandwidth requirements depend on the number of edges processed (examined), number of edges traversed (that pass the filter), and the number of vertices in the frontier over all iterations. For instance, an update to the list of visited vertices only happens if the edge actually passes the filter. Typically, the number of edges traversed is roughly equal to the permeability of the filter times the number of edges processed. To get a more accurate estimate, we collected statistics from one of the synthetically generated R-MAT graphs that are used in our experiments. These statistics are summarized in Table 3. Similarly, we quantify the volume of data movement by operation and memory access type (*random*, *stanza-like*, and *streaming*) noting the corresponding bandwidth on Mirasol, our Intel Xeon E7-8870 test system, in Table 4. Combining Tables 3 and 4, we calculate the average number of processed edges per second as a function of filter permeability by summing data movement time by type and inverting.

Figure 16 presents the resultant Roofline-inspired model for Mirasol. Note that these are all upper bounds on the best performance achievable and the underlying implementation might incur additional overheads from internal data structures, MPI buffers, etc. For example, it is common to locally sort the discovered vertices to efficiently merge them later in the incoming processor; an overhead we do not account for as it is not an essential step of the algorithm.

As the Roofline model selects ceilings by optimization, and bounds performance by their minimum, we too may select a filter implementation (pure Python KDT, SEJITS+KDT, or the CombBLAS limit) and the weighted bandwidth limit (in black) and look for the minimum.

We observe a pure Python KDT filter will result in a performance bound that is lower than the bandwidth limit. Conversely, the bandwidth limit is about 5× lower than the CombBLAS in-core performance limit. Ultimately, the performance of a SEJITS specialized filter is sufficiently fast to ensure a BFS implementation will be bandwidth-bound. This is a very important observation that explains why SEJITS+KDT performance is so close to CombBLAS performance in practice even though its in-core performance is about 2.3× slower.

Table 4: Breakdown of the volume of data movement by memory access pattern and operation.

Memory access type	Vertices visited	Edges traversed	Edges processed	Bandwidth on Mirasol
Random	24 bytes	8 bytes	0	9.09 GB/s
Stanza	0	0	24 bytes	36.6 GB/s
Stream	8 bytes	32 bytes	0	106 GB/s

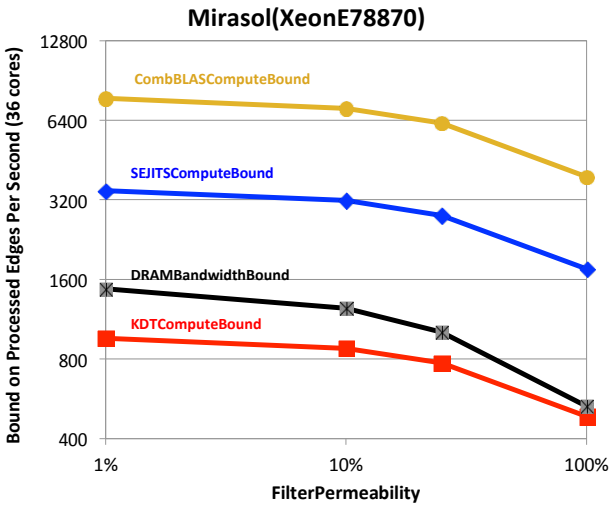


Figure 16: Roofline-inspired model for filtered BFS computations. Performance bounds arise from bandwidth, CombBLAS, KDT, or SEJITS+KDT filter performance, and filter success rate.

5.6 Discussion

DSELs and Runtime Code Generation. Structurally, the Asp framework provides two distinct sets of capabilities—DSL embedding and runtime code generation—that work particularly well when combined: DSELs with well-chosen abstractions allow capturing programmer intent rather than implementation, simplifying the task of generating good code, while runtime code generation allows the target code to make late decisions based on details of the target platform and even the input data for each problem instance. Indeed, the matrix powers ($\mathbf{A}^k\mathbf{x}$) specializer of Section 5.2 is not really a DSEL at all but a single method call, yet the characteristics of the problem (dependence on structure of input matrices, knowledge of cache geometry for tiling and blocking, etc.) made it appealing to use Asp's runtime code generation as the delivery mechanism for the specializer, and as an added benefit allowed the optimized code to be called from Python. Conversely, a separate specializer on which we have not reported in this paper generates Java code for Hadoop (an open source implementation of Map/Reduce), allowing Python programs to embed other specializers in the body of a Map/Reduce computation. In that case, the DSEL embedding was more useful than the ability to generate optimized code at runtime.

We conclude that the ability to embed lightweight DSELs and construct their compilers is separate from the ability to execute those compilers at runtime, but combining the two mechanisms opens new opportunities for bridging the productivity/performance gap.

Disadvantages of DSLs. There are two main disadvantages to a DSL. The first is that programmers must learn a new language with a new syntax. We address this problem by *embedding* the domain-specific languages in a common host language. DSELs ease integration with existing code and allow developers to use a familiar syntax, making them, to a developer, appear as simple as using libraries.

The second disadvantage is that the DSL implementor must invest substantial effort in tasks tangential to the problem domain, such as parsing the source language, performing generic optimizations, emitting the target language, and providing facilities for general computation that are outside the problem domain (file manipulation, e.g.). To this end, Asp is essentially a framework for doing JIT code generation of kernel-specific embedded DSLs, specifically designed to make it easy to capture a human-expert-created computation strategy.

Reusability and Extensibility. Beyond simply moving the complexity of getting good performance from one site in the application to another, specializers are easily usable in new Python applications, and some such as the GMM specializer can even transparently target either multicore CPU or GPU at runtime depending on hardware availability. Reuse can be increased by providing existing specializers with additional code-generation back ends (for new hardware or compilers) or additional DSEL front-ends (for exposing the specializer to other productivity languages).

While the most visible customers of Asp specializers are productivity programmers, Asp allows efficiency programmers who devise and optimize highly-performant code to encapsulate their strategy in a specializer, greatly increasing

the potential for flexible reuse of that strategy. Although the initial effort for writing a specializer is more work than a one-off optimization of a particular application, we argue that the benefit is far greater— many more people can take advantage of the knowledge of an efficiency-level programmer through a specializer, and the overall effort is less than if the efficiency-level programmer were to hand-optimize several instances of computations in the same domain.

Asp also provides an incremental adoption path for efficiency-level programmers: they can take an existing prototype written in an efficiency-level language, rapidly encapsulate it in an Asp specializer, and then gradually refine and generalize it over time. A common path is to embed an existing prototype in a template (see Section 4) and gradually add more "holes" into the template, into which runtime-computed values are substituted. Eventually an entire function or functions may be generated at runtime using tree transformation techniques. Automated tests help to ensure behavior is preserved during this refactoring. This incremental adoption strategy proved valuable in practice for our early adopters: the GMM and $\mathbf{A}^k\mathbf{x}$ specializer authors both began with a specific efficiency-level implementation that had limited support for sophisticated optimizations or multiple targets, and then added these features as the specializers were generalized.

Performance Portability. The SEJITS approach provides performance portability: if the specializer can generate code for the target platform (e.g. x86 multicore) then the same Python application will get high performance across many machines; if the specializer can target multiple platforms (e.g. GPUs as well as multicore) then the same application will get high performance even across platforms.

If no specializer exists, the code is still source-portable since it can run in unmodified Python. The portability aspect of productivity languages remains; with the SEJITS approach, that portability also extends across architectures and platforms.

Graph Algorithms.

The KDT graph analytics system achieves customizability through user-defined filters, high performance through the use of a scalable parallel library, and conceptual simplicity through appropriate graph abstractions expressed in a high-level language.

We have shown that the performance hit of expressing filters in a high-level language can be mitigated by Just-in-Time Specialization. In particular, we have shown that our embedded DSL for filters can enable Python code to achieve comparable performance to a pure C++ implementation. A roofline analysis shows that the specializer enables filtering to move from being compute-bound to memory bandwidth- bound. We demonstrated our approach on both real-world data and large generated datasets. Our approach scales to graphs on the order of hundreds of millions of edges, and machines with thousands of processors.

In future work we will further generalize our DSL to support a larger subset of Python, as well as expand SEJITS support beyond filtering to cover more KDT primitives. An open question is whether CombBLAS performance can be pushed closer to the bandwidth limit by eliminating internal data structure overheads.

6 Related Work

Going beyond libraries for domain experts. A popular way to provide good performance to productivity-language programmers has been to provide native libraries with high-level-language bindings such as SciPy (scipy.org), Biopython (biopython.org), BLAS [6], ScaLAPACK [5], and FFTW [15].

However, some DSEL benefits are difficult to achieve with libraries. One difficulty lies in conditioning code generation on the input problem parameters, as our GMM and $\mathbf{A}^k\mathbf{x}$ specializers do. The OSKI (Optimized Sparse Kernel Interface) sparse linear algebra library [38] precompiles 144 variants of each supported operation based on install-time hardware benchmarks that take hours to run, and includes logic to select the best variant at runtime, but applications using OSKI must still intermingle tuning code (hinting, data structure preparation, etc.) with the code that performs the calls to do the actual computations

Another difficulty is the frequent mismatch between the library's API and the preferred domain abstractions. For example, whereas solving a matrix in MATLAB is as simple as writing $X = A\backslash B$, the same operation in ScaLAPACK requires the application programmer to determine the processor layout, initialize array descriptors for each input and output matrix, load the data so that each processor has the correct portions of the input data, and finally call the solve function. A higher-level language could be layered over ScaLAPACK to reduce this impedance mismatch; indeed, our approach does just this, but takes the additional step of embedding the higher-level language into a common host language and allowing greater flexibility in how the language is JIT-compiled.

Finally, most widely-used libraries written in efficiency languages do not gracefully handle higher-order functions

as we needed for the stencil and graph specializers—even if the productivity language from which they're called does support them. This is usually because the efficiency languages do not have well-integrated support for higher order functions themselves. Even libraries such as Intel's Array Building Blocks (http://software.intel.com/en-us/articles/intel-array-building-blocks/) or others using C++ expression templates cannot benefit from all the runtime knowledge available to DSEL compilers. Because SEJITS allows JIT compilation *and* allows DSELs to leverage the host language's support of higher-order functions, it supports the tuning of such computations by lowering and inlining the interior functions. Thus, from the productivity programmer's point of view, the experience of using SEJITS DSELs is not only similar to a library, but idiomatic in the productivity language.

DSLs for bridging performance and productivity. Like the Delite project [9] (the most similar recent work to our own), we exploit domain-specific information available in a DSEL to improve compilation effectiveness. In contrast to that work, we allow compilation to external code directly from the structural and computational patterns expressed by our DSELs; in the Delite approach, DSELs are expressed in terms of lower-level patterns and it is those patterns that are then composed and code-generated for. Put another way, by eschewing the need for intermediate constructs, SEJITS "stovepipes" each DSEL all the way down to the hardware without sacrificing either domain knowledge or potential optimizations. The most prominent example of this is that SEJITS allows using efficiency-language templates to implement runtime-generated libraries (or "trivial" DSLs). Furthermore, auto-tuning is a central aspect of our approach, enabling high performance without needing complex machine and computation models.

The Weave subpackage of SciPy allows users to embed C++ code in strings inside Python code; the C++ code is then compiled and run, and uses the Python C API to access Python data. Cython (cython.org) is an effort to write a compiler for a subset of Python, while also allowing users to write extension code in C. Both of these expose low-level interfaces for efficiency programmers. Closer to our own approach is Copperhead [8], which provides a Python-embedded DSEL that translates data-parallel operations into CUDA GPU code.

Auto-tuning. The idea of using multiple variants with different optimizations is a cornerstone of auto-tuning. Auto-tuning was first applied to dense matrix computations in the PHiPAC library (Portable High Performance ANSI C) [4]. Using parameterized code generation scripts written in C, PHiPAC generated variants of generalized matrix multiply (GEMM) with a number of optimizations plus a search engine, to, at install time, determine the best GEMM routine for the particular machine. The technology has since been broadly disseminated in the ATLAS package (math-atlas.sourceforge.net). Auto-tuning libraries include OSKI (sparse matrix-vector multiplication) [38], SPIRAL (Fast Fourier Transforms) [34], and stencils [21, 35], in each case showing large performance improvements over non-autotuned implementations. With the exception of SPIRAL and Pochoir, all of these code generators use ad-hoc Perl or C with simple string replacement, unlike our template and tree manipulation systems.

Finally, our use of SEJITS to allow productivity-language (nonexpert) users to express graph computations in terms of simple filter predicates rather than converting the graph computation into a custom set of semiring algebra operators callable from CombBLAS has no analogue in Delite. (The Pegasus [22] framework is similar to CombBLAS, but its operations lack the algebraic completeness of CombBLAS's semiring framework.)

Specialization. Early work on specialization appeared in the Synthesis Kernel, in which a code synthesizer specialized kernel routines on-the-fly when possible [33]. Engler and Proebsting [13] illustrated the benefits of dynamically generating small amounts of performance-critical code at runtime. Jones [20, 17] and Thibault and Consel [37] proposed a number of runtime specialization transformations to increase performance of programs, including partial evaluation or interpreters customized for specific programs. Despite their different contexts, these strategies, like SEJITS, rely on selectively changing execution of programs using runtime as well as compile-time knowledge.

Just-in-time code generation and compilation. Sun's HotSpot JVM [31] performs runtime profiling to decide which functions are worth the overhead of JIT-ing, but must still be able to run arbitrary Java bytecode, whereas SEJITS does not need to be able to specialize arbitrary productivity-language code. Our approach is more in the spirit of Accelerator [36], which focuses on optimizing specific parallel kernels for GPU's while paying careful attention to the efficient composition of those kernels to maximize use of scarce resources such as GPU fast memory. Asp is more general in allowing specializer writers to use the full set of Python features to implement specializers that capture their problem-solving strategy.

Ongoing work. To support auto-tuning, we are building a global database that will aggregate runtime information (performance, hardware characteristics, and problem parameters) from specializers running at different sites. These data will enable future specializer invocations to base tuning parameters on information from similar problems on similar machines. As with specializers that take advantage of the input problem data, there is a tradeoff between the efficiency gained by using such information and the cost of repeating one or more specializer phases when the inputs are altered

in a way that affects the result. We intend to investigate this tradeoff using modeling techniques based on machine learning.

We are investigating how best to compose specializers. We believe many combinations of Semantic Models can be composed while preserving both their productivity-level abstractions and the efficiency-level performance of the composition. Like database query optimization [2], we have a tree of operators communicating over edges, each of which may afford multiple implementations. Concepts from parallel/distributed query optimization, including independent parallelism, pipelined parallelism, partitioned parallelism [12] and adaptive query processing [3] could be applied to help allocate parallel resources to specializers and select strategies that allow specializers to cooperate effectively.

Our prototype SEJITS framework uses Python as both the embedding language and the specializer implementation language, but these tasks are logically separate. Any modern scripting language that supports introspection, dynamic linking at runtime, and has a reasonable foreign function interface will serve as an embedding language. Ruby, Scala (http://scala.org) and Lua (http://lua.org), for example, all have these properties.

Errors in a complex specializer, manifesting as incorrect behavior of generated code or cryptic feedback to the application programmer, can be difficult to diagnose due to its dynamic and language-crossing nature. Tool support for reproducing and isolating errors, visualizing and verifying code transformations, producing clear error messages, and modular testing of specializers is essential.

Finally, we have only begun building specializers for a few domains. Other specializers, especially those expressing higher-level patterns such as Map, are in development as a response to application needs. We are also investigating other target platforms for specializers such as public cloud computing.

7 Conclusions

We have attempted to support two claims applying to two different types of SEJITS stakeholders. For *efficiency programmers* accustomed to writing high-performance code, SEJITS and the Asp library make their work more easily available and widely reusable by productivity programmers than would a standalone compiled library. For *productivity programmers*, SEJITS provides both parallel efficiency-level performance and performance portability across hardware, all with the same effort as sequential productivity programming.

Because each specializer is essentially a self-contained mini-compiler for a particular DSEL, new specializers can be continuously added to the ecosystem. We hope others will contribute to this ecosystem and help raise the level of abstraction for productivity programmers while taking advantage of state-of-the-art efficiency-language parallel code.

The software created in this project is available at sejits.org. It relies on other open source software including Python, SciPy, NumPy, and CodePy.

References

[1] X Anguera, Simon Bozonnet, Nicholas W D Evans, Corinne Fredouille, G Friedland, and O Vinyals. Speaker diarization : A review of recent research. *Accepted for publication in "IEEE Transactions On Acoustics Speech and Language Processing" (TASLP), special issue on "New Frontiers in Rich Transcription", 2011*, 2011.

[2] P.M.G. Apers, A.R. Hevner, and S.B. Yao. Optimization algorithms for distributed queries. *Software Engineering, IEEE Transactions on*, SE-9(1):57 – 68, Jan. 1983.

[3] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proceedings of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00, pages 261–272, New York, NY, USA, 2000. ACM.

[4] Jeff Bilmes, Krste Asanović, Chee-Whye Chin, and Jim Demmel. Optimizing matrix multiply using PHiPAC: a Portable, High-Performance, ANSI C coding methodology. In *Proceedings of International Conference on Supercomputing*, Vienna, Austria, July 1997.

[5] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, Philadelphia, 1997. www.netlib.org/scalapack.

[6] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of Basic Linear Algebra Subroutines (BLAS). *ACM Trans. Math. Soft.*, 28(2), June 2002. www.netlib.org/blas.

[7] Aydın Buluç and John R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. Technical Report UCSB-CS-2010-18, University of California, Santa Barbara, 2010.

[8] Bryan Catanzaro, Shoaib Kamil, Yunsup Lee, Krste Asanović, James Demmel, Kurt Keutzer, John Shalf, Kathy Yelick, and Armando Fox. SEJITS: Getting productivity and performance with selective embedded JIT specialization. In *Workshop on Programming Models for Emerging Architectures (PMEA 2009)*, Raleigh, NC, October 2009.

[9] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. In Calin Cascaval and Pen-Chung Yew, editors, *PPOPP*, pages 35–46. ACM, 2011.

[10] Henry Cook, Ekaterina Gonina, Shoaib Kamil, Gerald Friedland, and David Pattersonand Armando Fox. Cuda-level performance with python-level productivity for gaussian mixture model applications. In *3rd USENIX conference on Hot topics in parallelism (HotPar'11)*, Berkeley, CA, USA, 2011.

[11] Kaushik Datta. *Auto-tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, EECS Department, University of California, Berkeley, Dec 2009.

[12] Amol Deshpande and Lisa Hellerstein. Flow algorithms for parallel query optimization. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 754–763, Washington, DC, USA, 2008. IEEE Computer Society.

[13] Dawson R. Engler and Todd A. Proebsting. Dcg: an efficient, retargetable dynamic code generation system. In *ASPLOS 1994*, pages 263–272, New York, NY, USA, 1994. ACM.

[14] Paul Erdős and Alfréd Rényi. On random graphs. *Publicationes Mathematicae*, 6(1):290–297, 1959.

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. Special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 361–366, New York, NY, USA, 2005. ACM.

[17] Robert Glück and Neil D. Jones. Automatic program specialization by partial evaluation: an introduction. In *Software Engineering in Scientific Computing*, pages 70–77, 1996.

[18] Paul Hudak. Building domain-specific embedded languages. *ACM Comput. Surv.*, 28:196, December 1996.

[19] NVIDIA Inc. CUDA parallel programming and computing platform. http://www.nvidia.com/object/cuda_home_new.html.

[20] Neil D. Jones. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52:307–339, August 2004.

[21] Shoaib Kamil, Cy Chan, Leonid Oliker, John Shalf, and Samuel Williams. An auto-tuning framework for parallel multicore stencil computations. In *IPDPS'10*, pages 1–12, 2010.

[22] U. Kang, C.E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations. In *ICDM'09. IEEE Int. Conference on Data Mining*, pages 229–238. IEEE, 2009.

[23] H. Kwak, C. Lee, H. Park, and S. Moon. What is Twitter, a social network or a news media? In *WWW '10: Proceedings of the 19th international conference on World wide web*, pages 591–600, New York, NY, USA, 2010. ACM.

[24] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, Mathematically Tractable Graph Generation and Evolution, Using Kronecker Multiplication. In *PKDD*, pages 133–145, 2005.

[25] M. Luby. A simple parallel algorithm for the maximal independent set problem. In *Proceedings of the seventeenth annual ACM symposium on Theory of computing*, STOC '85, pages 1–10, New York, NY, USA, 1985. ACM.

[26] A. Lugowski, D. Alber, A. Buluç, J. Gilbert, S. Reinhardt, Y. Teng, and A. Waranis. A Flexible Open-Source Toolbox for Scalable Complex Graph Analysis. In *SDM'12*, pages 930–941, April 2012.

[27] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. http://www.cs.virginia.edu/stream/.

[28] K.C. McPhee, C. Denk, Z. Al Rekabi, and A. Rauscher. Bilateral filtering of magnetic resonance phase images. *Magn Reson Imaging*, 2011.

[29] Marghoob Mohiyuddin, Mark Hoemmen, James Demmel, and Kathy Yelick. Minimizing communication in sparse matrix solvers. In *Supercomputing 2009*, Portland, OR, Nov 2009.

[30] Francisco Ortigosa, Mauricio Araya-Polo, Felix Rubio, Mauricio Hanzich, Raud de la Cruz, and Jose Maria Cela. Evaluation of 3d rtm on hpc platforms. *SEG Technical Program Expanded Abstracts*, 27(1):2879–2883, 2008.

[31] Michael Paleczny, Christopher Vick, and Cliff Click. The Java HotSpot server compiler. In *Java Virtual Machine Research and Technology Symposium (JVM '01)*, April 2001.

[32] Andrew D. Pangborn. Scalable data clustering using gpus. Master's thesis, Rochester Institute of Technology, 2010.

[33] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1:11–32, 1988.

[34] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[35] Yuan Tang, Rezaul Alam Chowdhury, Bradley C. Kuszmaul, Chi-Keung Luk, and Charles E. Leiserson. The

Pochoir stencil compiler. In *Proceedings of the 23rd ACM symposium on Parallelism in algorithms and architec- tures*, SPAA '11, pages 117–128, New York, NY, USA, 2011. ACM.

[36] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: Using data parallelism to program gpus for general-purpose uses. In *ASPLOS 2006*, pages 325–335, 2006.

[37] Scott Thibault, Charles Consel, Scott Thibault Charles Consel, Julia L. Lawall, and Renaud Marlet Gilles Muller. Static and dynamic program compilation by interpreter specialization. In *Higher-Order and Symbolic Computation*, pages 161–178, 2000.

[38] R Vuduc, J W Demmel, and K A Yelick. Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics Conference Series*, 16(i):521–530, 2005.

[39] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[40] Samuel Williams, Andrew Waterman, and David A. Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, pages 65–76, 2009.

[41] David Wonnacott. Using time skewing to eliminate idle time due to memory bandwidth and network limitations. *Parallel and Distributed Processing Symposium, International*, 0:171, 2000.

[42] J. Yang and J. Leskovec. Patterns of temporal variation in online media. In *Proceedings of the fourth ACM international conference on Web search and data mining*, WSDM '11, pages 177–186, New York, NY, USA, 2011. ACM.

List of Symbols, Abbreviations and Acronyms

Akx or $A^k x$	the Matrix Powers kernel used in some Krylov-subspace solvers
API	Application Programming Interface
Asp	An implementation of SEJITS (q.v.) for Python
BFS	Breadth-first search of a graph
BLAS	Basic Linear Algebra Subprograms
CG	Conjugate gradient, a method for solving sparse linear systems
CodePy	A library for compiling and managing C functions called from Python
CUDA	the Compute Unified Device Architecture parallel programming framework
DSEL	Domain-specific embedded language
DSL	Domain-specific language
EM	Expectation-maximization method for training a model to match a set of observed training data
FFI	Foreign function interface, a way of calling functions compiled in one language from code written in another
gcc	The GNU C Compiler
GMM	Gaussian mixture model
GPU	Graphics processing unit
KDT	Knowledge Discovery Toolkit
MD5	A one-way hash function for creating digests of large objects that avoids hash collisions with high probability
MIS	maximal independent subset of a graph such that no two vertices of the subgraph are connected to each other and all other vertices outside MIS are connected to at least one member of the MIS
MKL	the optimized Math Kernel Library supplied by Intel Corp.
NumPy	A third-party Python library that provides for efficient numeric-array data structures that can be shared between Python and C/C++ code
OpenMP	An open standard for parallel processing that works by annotating C code with pragmas identifying data-parallel constructs that can be exploited by a compatible compiler
PHiPAC	Portable High-Performance ANSI C, an early autotuner
POSIX	An ANSI standard describing the programming interfaces of Unix-like systems
RMAT	Realistic, Mathematically Tractable Graph Generation and Evolution Using Kronecker Multiplication [24]
Roofline	a visually intuitive representation of performance bounds on a particular kernel arising from bandwidth or compute limits on a particular architecture [39]
SAID	semiring additive identity operation
SIMD	Single instruction, multiple datastream parallel computing
Scala	A byte-compiled programming language featuring dynamic type inference
SciPy	The standard library for scientific computing operations included with the Python language
SEJITS	Selective, Embedded, Just-in-Time Specialization
SpGeMM	Sparse generalized matrix-matrix multiplication
SPIRAL	A DSL and library for Fast Fourier Transforms [34]
SpMV	Sparse matrix-vector multiplication
STREAM	(Not an acronym) A benchmark for sustainable memory bandwidth in high-performance computers [27]